Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

Ambienti Operativi: Script Bash

Amos Brocco, Ricercatore, DTI / ISIN

Script Bash

- Gli script sono dei file contententi dei comandi che vengono eseguiti dalla shell in sequenza

script.sh

```
#!/bin/bash

# Questo è un commento, e inizia con il cancelletto
echo “Ciao mondo”
```

La sequenza che inizia con **#!** (in gergo sha-bang) determina l'interprete da utilizzare per eseguire lo script

- Uno script può essere eseguito richiamando l'interprete oppure direttamente (se l'utente ha privilegi di esecuzione sul file)

Variabili

- Le variabili funzionano come in qualsiasi altro linguaggio di programmazione.
- Le variabili possono rappresentare delle stringhe, valori numerici interi, oppure degli array

script.sh

```
#!/bin/bash

i=4
x="ciao mondo"

echo $x

l=(A B C D)

echo ${l[0]} ${l[2]}
```

Niente spazi intorno a = (riferito a `i=4`)

Per ottenere il valore di una variabile bisogna utilizzare il simbolo **$** (riferito a `echo $x`)

Variabili

- L'operatore **$** “espande” il valore della variabile (ovvero sostituisce all'espressione il valore associato alla variabile)
  - l'espansione di una variabile ha la precedenza sul globbing:

script.sh

```
#!/bin/bash

template=”Immagine”
ls $template???.jpg
ls $template[[:digit:]]??.jpg
```

- Le variabili possono trovarsi anche all'interno di stringhe:
  - se la stringa è delimitata da “ “ le variabili verranno espanse
  - per evitare espansione utilizzare **\$** (escaping)

script.sh

```
#!/bin/bash
messaggio=”Ciao mondo”
echo “Computer dice $messaggio”
```

Variabili

- Parametri posizionali passati allo script:
  - La variabile **$0** corrisponde al nome dello script
  - Le variabili **$1 .. $n** sono gli altri parametri
- Numero dei parametri: **$#**
- Tutti i parametri: **$@**
- La variabile **$$** contiene il PID del processo della shell corrente
- La variabile **$!** contiene il PID dell'ultimo processo mandato in background
- La variabile **$?** corrisponde al valore di ritorno dell'ultimo comando eseguito

- Uno script può terminare e ritornare un valore di ritorno con **exit**

script.sh

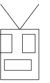
```
#!/bin/bash
if [ $# -ne 1 ] then
            echo "Utilizzo: $0 pianeta"
            exit 0
fi

echo “Hello $1”
```

Declare

- Con declare è possibile definire o impostare una variabile a un tipo specifico

```
declare opzione nomevariabile[=valore]
```



-a	**Array**
-A	**Array associativo**
-i	**Tipo intero**
-l	**Stringa in minuscolo**
-r	**Variabile in sola lettura**
-u	**Stringa in minuscolo**
-x	**Variabile esportata**

Shift

- Il comando shift sposta i valori dei parametri posizionali
  - Posso specificare un parametro *n* per indicare di quante posizioni spostarmi

```
[X] bash
utente@host:~$ ./mioscript.sh “Ciao” “mondo” 100 200 300
```

Ciao	mondo	100	200	300
$1	$2	$3	$4	$5

shift

~~Ciao~~	mondo	100	200	300	
	$1	$2	$3	$4	~~$5~~

Array

- Un array è una lista statica di elementi, può essere definito con (...) oppure inserendo degli elementi con `[...]`
- La lunghezza di un array è:
  - `${#array[@]}`
- Per accedere agli elementi
  - `${array[indice]}`
- Gli indici iniziano a **0**

script.sh

```
#!/bin/bash
array=(a b c d e f)

echo $array
echo "Primo elemento ${array[0]} "
echo "Quarto elemento ${array[3]}"
echo "Tutti gli elementi ${array[@]}"
echo "Dimensione ${#array[@]} "

altroarray[0]=”Una stringa”
```

Stampa il primo elemento!

Array

- Slicing: estrarre una porzione di un array

script.sh

```
#!/bin/bash
array=(a b c d e f)

echo “b c d e slice ${array[@]:1:4}”
echo “Da b alla fine ${array[@]:1}”
echo "Fino a c ${array[@]::3}"
```

Il primo valore è l'indice di inizio. Se omesso, l'indice è 0.

Il secondo valore (se presente) è il numero di elementi

Array

- Fusione di due array:

script.sh

```
#!/bin/bash
alpha=(a b c d e f)
beta=(p q r s)

gamma=( ${alpha[@]} ${beta[@]} )
```

Utilizzando lo slicing possiamo inserire / togliere degli elementi da un array

Array associativi

- Gli array associativi possono contenere valori indicizzati con chiavi alfanumeriche

script.sh

```
#!/bin/bash
declare -A arrayassoc

arrayassoc[uno]=Ciao
arrayassoc[due]=Mondo


echo ${arrayassoc[uno]}
```

Non necessario, l'array è creato quando inseriamo il primo valore

Associamo le chiavi 'uno' e 'due' ai valori 'Ciao', rispettivamente 'Mondo'

script.sh

```
echo ${arrayassoc[@]}
echo ${#arrayassoc[@]}
```

Possiamo leggere tutti gli elementi e il numero di elementi nello stesso modo di un array tradizionale

Input utente

- Con il comando **read** è possibile leggere da *stdin* l'input dell'utente e memorizzarlo in una variabile

script.sh

```
#!/bin/bash
echo “Come ti chiami?”
read nome
echo “Ciao $nome!”

read -p “Come ti chiami?” nome
echo “Ciao $nome!”
```

Il parametro **-p** permette di specificare un messaggio di prompt da visualizzare all'utente

Espressioni aritmetiche

- Per valutare le espressioni aritmetiche possiamo utilizzare
  - il comando **expr**
  - la forma **$(( ))** ($ può essere omesso nei test)
  - l'espressione **let**

script.sh

```
#!/bin/bash
i=4

j=`expr $i '*' 2`
j=`expr $j + 1`

j=$(( j + 3 ))

let j=j+1
```

Devo usare le virgolette semplici ''
per evitare il globbing di *

If ... then ... else

Sintassi if .. then

```
if condizione then
        ...
fi
```

Sintassi if .. then .. else

```
if condizione then
    ...
else
    ...
fi
```

Sintassi if .. then .. elif .. else

```
if condizione then
    ...
elif condizione2 then
        ...
else
    ...
fi
```

Condizionali

- bash utilizza il comando **`test`**, abbreviato con **`[ ]`**
  - es. **`[ $x -eq $y ]`** equivale a **`test $x -eq $y`**
- Un'espressione 'vera' (true) ritorna un codice di uscita **0**

Possono essere sostituiti da <,>,<=,>= utilizzando il test di espressioni aritmetiche (( ))

-n *op*	Stringa op non ha lunghezza 0
-z *op*	Stringa op ha lunghezza 0
-d *op*	Esiste una directory op
-e op	Esiste il file op
a -eq b	a e b sono numeri interi e sono uguali
a -neq b	Opposto di -eq
a = b	a è uguale a b (come stringa)
a != b	Opposto di =
a -lt b	a < b (a, b numeri interi)
a -gt b	a > b (a, b numeri interi)
a -le b	a <= b (a, b numeri interi)
a -ge b	a >= b (a,b numeri interi)
a -a b	AND logico
a -o b	OR logico
! a	NOT logico

Guida ai condizionali: help \[\[oppure help test

Differenza tra [ ] e [[ ]]

- Per delimitare i condizionali bash supporta
  - **[ ]** singole parentesi quadre (brackets)
  - **[[ ]]** doppie parentesi quadre
- Le singole parentesi sono “più vecchie” e garantiscono una maggior compatibilità con altre shell
- Le doppie parentesi sono “più nuove” e semplificano l'utilizzo di alcuni costrutti
  - Non bisogna fare l'escaping di **>** e **<**
  - **&&** sostituisce -a (AND logico)
  - **||** sostituisce -o (OR logico)

Differenza tra [ ] e [[ ]]

Il > (maggiore) tra stringhe ritorna una condizione di uscita 0 (vero) se il valore ASCII della prima stringa è più grande di quello della seconda

[X] bash

```
utente@host:~$ if [ "a" > "b" ]; then echo "Questo non deve essere stampato"; fi
Questo non deve essere stampato
utente@host:~$ if [ "a" \> "b" ]; then echo "Questo non deve essere stampato"; fi
utente@host:~$ if [ "c" \> "b" ]; then echo "Questo deve essere stampato"; fi
Questo deve essere stampato
utente@host:~$ if [[ "a" > "b" ]]; then echo "Questo non deve essere stampato"; fi
utente@host:~$
```

Condizionali

script.sh

```
X=2
Y=10

if [ $X -lt $Y ]; then
	echo “$X è minore di $Y”
elif [ $X = $Y ]
then
	echo “$X è uguale a $Y”
else
	echo “$X è maggiore di $Y”
fi
```

Si può omettere il punto e virgola solo se 'then' è in una nuova riga

script.sh

```
A=””
if [ -n $A ]; then
	echo "La stringa non è vuota (1)"
fi

if [ -n "$A" ]; then
	echo "La stringa non è vuota (2)"
fi
```

Non funziona, perché $A **viene espansa prima del test**, quindi il condizionale diventa: `test -n`

Per il test delle stringhe utilizzare sempre le virgolette attorno alle variabili!

Condizionali

- bash implementa anche altri costrutti per i test
  - http://tldp.org/LDP/abs/html/testconstructs.html#DBLBRACKETS
- La forma aritmetica ((...)) (senza $) può essere usata per i test
  - http://tldp.org/LDP/abs/html/dblparens.html

While

Sintassi

```
while condizione; do
          ...
done
```

Esempio

```
i=1

while [ $i -le 10 ]; do
          echo “il valore di i è $i”
          i=`expr $i + 1`
done
```

Si può omettere il punto e virgola solo se 'do' è in una nuova riga

Esempio di utilizzo del while

- Scorrere la lista dei parametri

script.sh
```
#!/bin/bash

while [[ $# > 0 ]]; do
            echo $1
            shift
done
```

Until

- Esegue il ciclo fintanto che la condizione non è vera

Sintassi

```
until condizione; do
        ...
done
```

Esempio

```
contatore=10
until [ $contatore -lt 1 ]; do
        echo $contatore
        let contatore-=1
done
```

For

- 2 varianti

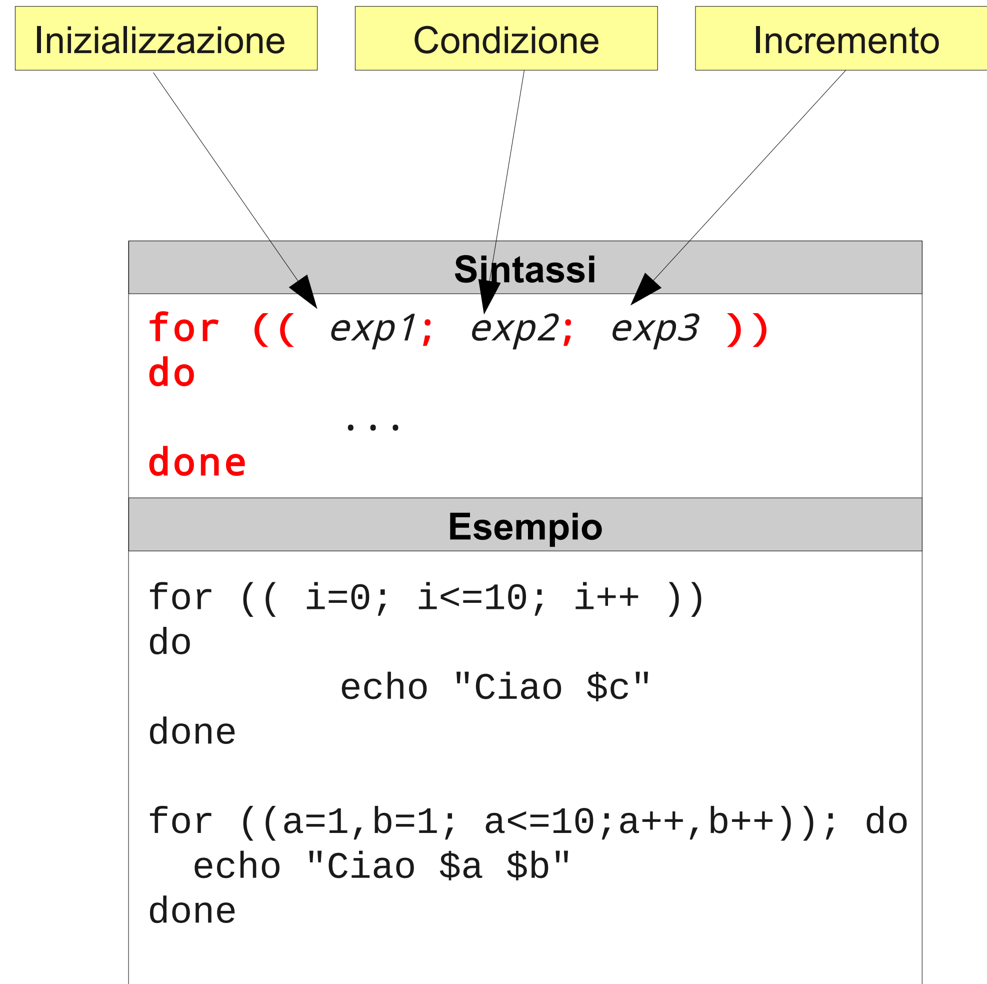
Sintassi

```
for nomevar in elementi; do
        ...
done
```

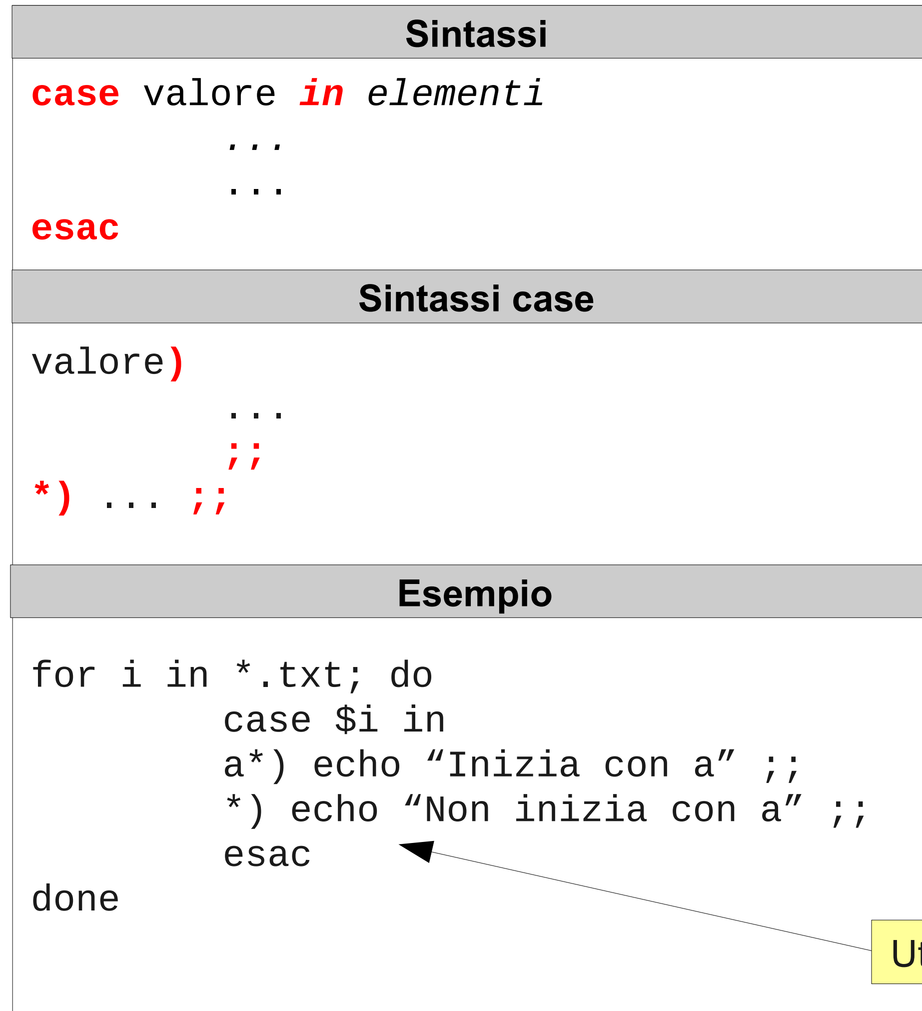
Esempio

```
for i in *.txt; do
        echo $i
done


for i in A B C; do
        echo $i
done
```



Switch / Case



Break

- Per uscire da un ciclo possiamo utilizzare il comando **`break`**

script.sh

```
#!/bin/bash

while [[ $# > 0 ]]; do
            echo $1
            if [ “$1” = “x” ]; then
                        break;
            fi
            shift
done
```

Funzioni

Sintassi

```
function nome() {
        ...
}
```

è possibile tralasciare function oppure (), mai entrambi

Esempio

```
function saluta {
        echo "Ciao mondo"
}

saluta
```

Valore di ritorno è numero intero

```
somma() {
        return $(( $1 + $2 ))
}
somma 3 4
risultato=$?
```

Valore di ritorno (generico)

```
function moltiplica() {
    echo $(( $1 * $2 ));
}
echo $(moltiplica 4 3)
```

Here document

- Permette di redirigere una stringa (tipicamente su più righe) come input (stdin) a un comando con **`<<TAG`**

script.sh

```
cat <<MIOTAG
Questa è la stringa che
verrà scritta su schermo
da cat
MIOTAG
```

- La stringa è delimitata da un tag iniziale (dopo <<) e termina quando il tag si trova su una linea senza altro testo
- È possibile utilizzare la forma **`<<-TAG`** per eliminare i tabulatori all'inizio delle righe di input
- La forma **`<<"TAG"`** evita l'espansione all'interno della stringa

Funzionalità avanzate

Il simbolo $ non va messo davanti al nome della variabile

`${var:-parola}`	Se var esiste e non è vuota ritorna il suo valore, altrimenti ritorna parola
`${var:=parola}`	Se var esiste e non è vuota ritorna il suo valore, altrimenti imposta var a valore e ritorna valore
`${var:?messaggio}`	Se var esiste e non è vuota, ritorna il suo valore, altrimenti stampa “var:” seguito dal messaggio e termina l'esecuzione
`${var:+parola}`	Se var esiste e non è vuota, ritorna parola; altrimenti ritorna una stringa vuota
`${var:offset:l}`	Ritorna la sottostringa della lunghezza l specificata di var a partire dall'offset. Se lunghezza non è specificata, continua fino alla fine

Funzionalità avanzate

[X] bash

```
#!/bin/bash

toto="Ciao mondo"

# Se var esiste e non è vuota ritorna il suo valore, altrimenti ritorna parola
echo ${toto:-"Non esiste"} # Questo stampa "Ciao mondo"
echo ${teto:-"Non esiste"} # Questo stampa "Non esiste"

# Se var esiste e non è vuota ritorna il suo valore, altrimenti imposta var a valore e ritorna valore
echo ${toto:="Non esiste"} # Questo stampa "Ciao mondo"
echo ${teto:="Non esiste, creo variabile"} # Questo stampa "Non esiste, creo variabile", e crea teto
echo $teto

# Se var esiste e non è vuota, ritorna il suo valore, altrimenti stampa “var:” seguito dal messaggio e
# termina l'esecuzione
echo ${toto:?"non esiste"} # Questo stampa "Ciao mondo"
echo ${tito:?"non esiste"} # Questo stampa un errore e termina lo script

# Se var esiste e non è vuota, ritorna parola; altrimenti ritorna una stringa vuota
echo ${toto:+"Toto esiste"} # Questo stampa "Toto esiste"
echo ${tuto:+"Tuto esiste"} # Questo stampa una stringa vuota

# Ritorna una sottostringa di lunghezza 6 a partire dalla posizione 0
echo ${toto:0:6} # Questo stampa "Ciao m"
```

Funzionalità avanzate

`${var#pattern}`	Se pattern coincide con l'inizio di var rimuovi la parte più corta che coincide
`${var##pattern}`	Se pattern coincide con l'inizio di var, elimina al parte più lunga che coincide
`${var%pattern}`	Se pattern coincide con la fine di var, elimina la più corta parte che coincide
`${var%%pattern}`	Se pattern coincide con la fine di var, elimina la più lunga parte che coincide
`${var/pattern/str}`	La parte più lunga di var che coincide con pattern è sostituita da str
`${var//pattern/str}`	Tutte le volte che pattern coincide con una parte di var, sostituisci con str

È possibile combinare pattern matching con il globbing, es. `${x%.txt}` rimuove l'estensione .txt

Funzionalità avanzate

[X] bash

```
#!/bin/bash

corso="Ambienti operativi"

# Se pattern coincide con l'inizio di var rimuovi la parte più corta che coincide
echo ${corso#Amb*} # La parte più corta è Amb, stampa "ienti operativi"

# Se pattern coincide con l'inizio di var rimuovi la parte più lunga che coincide
echo ${corso##Amb*} # Stampa la stringa vuota!

# Se pattern coincide con la fine di var rimuovi la parte più corta che coincide
echo ${corso%*ivi} # La parte più corta che coincide è ivi, stampa "Ambienti operat"

# Se pattern coincide con la fine di var rimuovi la parte più lunga che coincide
echo ${corso%%*ivi} # Stampa la stringa vuota!

# La parte più lunga di var che coincide con pattern è sostituita da str
echo ${corso/i/o} # Stampa "Amboenti operativi"

# Tutte le volte che pattern coincide con una parte di var, sostituisci con str
echo ${corso//i/o} # Stampa "Amboento operatovo"
```